The Java HotSpot™ Virtual Machine, v1.4.1, d2

A Technical White Paper
September 2002

Table of Contents

Introduction and Overview 1
New in Java HotSpot Virtual Machine Release 1.4.1 2
Java HotSpot VM — Built on a Solid Foundation 3
Scalability 3
Performance 3
Reliability, Availability, Serviceability (RAS) 4
Earlier Enhancements 5
The Java HotSpot VM Architecture 6
Overview 6
Memory Model 7
Garbage Collection 8
64-bit Architecture 11
Ultra-Fast Thread Synchronization 12
New I/O APIs 13
Debugging Support 14
Overview 14
Java Virtual Machine Debugger Interface (JVMDI) 14
Java Debug Wire Protocol (JDWP) 15
Java Debug Interface (JDI) 15
Full-Speed Debugging 15
HotSwap Class File Replacement 15
VMDeathRequests 16
Logging 16
The Java HotSpot Compilers 17
Overview 17
Hot Spot Detection 18
Method Inlining 18
Dynamic Deoptimization 18
Java HotSpot Client Compiler 19
Java HotSpot Server Compiler 19
Impact on Software Reusability 20
Overview 20
Summary 21
Availability 22
Resources 23

Chapter 1

Introduction and Overview

Java HotSpot™ technology delivers higher performance and greater reliability over previous versions. Both client and server versions feature a number of improvements that enable developers to create more demanding applications in less time. Java HotSpot technology provides the foundation for the Java™ 2 Platform, Standard Edition (J2SE™) version 1.4.1 software, the premier solution for rapidly developing and deploying business-critical enterprise applications. J2SE technology is available for Microsoft Windows, Linux, and the Solaris™ Operating Environment (OE), as well as other platforms through Java technology licensees.

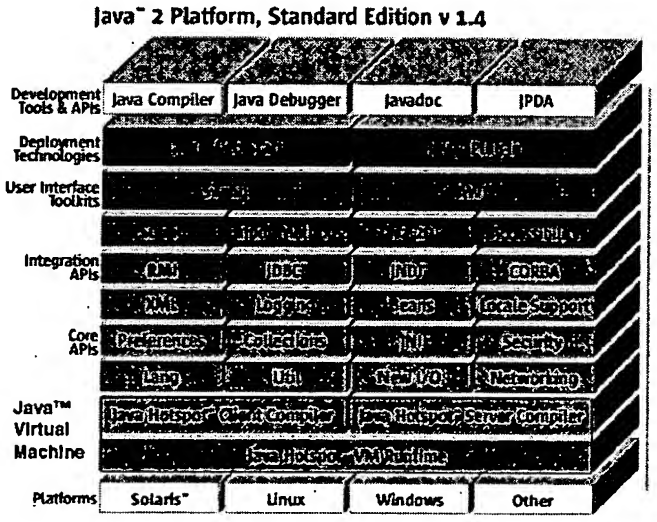

Figure 1-1: The Java HotSpot Virtual Machine is the foundation for the Java 2 Platform, Standard Edition technology.

The Java 2 platform has become a mainstream vehicle for software development and deployment. With millions of developers and users, the Java platform is growing explosively in many

dimensions: from credit cards to wireless devices, desktops to mainframes. It is the underlying foundation for deploying Web page applets, Web services, large commercial applications, and more.

This latest version of the Java HotSpot virtual machine (VM) builds upon Java technology's cross-platform support and robust security model with new features and capabilities for scalability, quality, and performance. In addition to new features, this version is upward compatible with previous releases.

The Java HotSpot VM supports virtually all aspects of development, deployment, and management of corporate applications, and is used by:

- Integrated development environments (IDEs) including the Sun™ Open Net Environment (Sun ONE) Developer Studio, Borland JBuilder, WebGain Visual Cafe, Oracle JDeveloper, Metrowerks CodeWarrior, and the NetBeans™ open source project.
- Application server vendors, such as the Sun ONE Application Server and BEA Systems WebLogic software.

Sun Microsystems, Inc. is also driving performance improvements through the use of various industry standard and internally developed benchmarks. These improvements are applicable to both the client- and server-side Java VM environments.

The Java 2 SDK, Standard Edition, contains two implementations of the Java VM:

- Java HotSpot Client VM, which is the default virtual machine of the Java 2 SDK and Java 2 Runtime Environment. It is tuned for best performance when running applications in a client environment by reducing application start-up time and memory footprint.
- Java HotSpot Server VM, which is designed for maximum program execution speed for applications running in a server environment.

New in Java HotSpot Virtual Machine Release 1.4.1

Java HotSpot 1.4.1 incorporates new features over previous versions of the Java HotSpot Virtual Machine. These include:

- The full-speed debugging feature of the Java Platform Debugger Architecture now can be used with both the Java HotSpot Client VM and Java HotSpot Server VM. In version 1.4.0, it worked only with the Java HotSpot Client VM. Full-speed debugging enables using Java HotSpot technology — instead of running in interpreter mode — when debugging programs.
- New garbage collection algorithms are available. Two new garbage collectors (GC) can now be used to improve application performance. The Java HotSpot memory system offers the flexibility to use specific GC algorithms according to the needs of individual applications. In addition to the GC mechanisms available in earlier releases, this latest version of the Java HotSpot VM offers a multithreaded synchronous copying collector which works well in many multiprocessor environments, and a mostly concurrent mark-sweep collector, which improves performance in applications that require large heaps.
- Beginning with J2SE 1.4.1, the Java HotSpot Server VM does not support operation on chips with SPARC™ v8 architecture. The SPARCstation™ family of processors, including the SPARCstation Workstation, SPARCstation Classic, SPARCstation 2, SPARCstation 4, SPARCstation 5, SPARCstation 10, SPARCstation 20, and SPARCstation Voyager™ processors, are affected by this change. The Java HotSpot Client VM does support operation on SPARC v8 platforms. See the Java HotSpot VM documentation for information on the Server VM and Client VM. Note that UltraSPARC™ processors are not affected by this change.

Java HotSpot VM — Built on a Solid Foundation

This version of the Java HotSpot VM builds on a strong foundation of features and capabilities. The previous version incorporated many improvements in scalability, performance, reliability, and serviceability (RAS). Overall performance was significantly enhanced by using a variety of techniques, as listed below. As well, on-the-fly adaptive optimization technology, which detects and accelerates performance-critical code, has been further refined and tuned. The main areas of improvement in version 1.4.0 include:

Scalability

- 64-bit support on the Solaris OE, SPARC Platform Edition: 64-bit support provides Java technology developers with near-limitless amounts of memory for high-performance, high-scalability computing. While earlier J2SE releases were limited to addressing four gigabytes of RAM, version 1.4.0 allows applications built on Java technology to access hundreds of gigabytes of RAM. This enables developers to drive more applications and very large datasets into memory, and avoids the performance overhead of reading data in from a disk or database. The result is significantly faster processing for business intelligence, data mining, and engineering and scientific applications. Developers can take advantage of the benefits of 64-bit computing without rewriting their existing Java technology-based applications. Note that 64-bit functionality is available only on Java HotSpot Server Compiler environments and the Solaris OE.

Performance

- Faster Reflection: The JVM now generates bytecode stubs for frequently used reflective objects such as Methods, Constructors, and Classes. This provides significant speedup in reflection-intensive code, such as that used in serialization. The improvements are visible when running RMI, Java Native Interface (JNI), or CORBA code environments.
- Object Serialization: The JVM also speeds up code paths in object serialization, specifically in accessing and setting fields, method invocation, and internal synchronization overhead. As with the reflection improvements, this is visible when running RMI, JNI, or CORBA code environments.
- New I/O (NIO): The NIO APIs introduced in version 1.4 provide features and improved performance in the areas of buffer management, scalable network and file I/O, character-set support, and regular-expression matching. For example, the requirement for one thread per network connection was removed. Reducing thread overhead enables more network connections. The NIO APIs supplement the I/O facilities in the `java.io` package. NIO introduced in version 1.4 is a subset of the changes proposed in JSR 51. Other NIO-related improvements include:
  - The NIO-related entry points allow native code to access `java.nio` `direct` buffers. The contents of a direct buffer can, potentially, reside in native memory outside of the ordinary garbage-collected heap. This allows memory-mapped data to be shared across the Java native boundary, resulting in dramatic performance improvements for certain operations.
  - An invocation interface routine, `AttachCurrentThreadAsDaemon`, allows native code to attach a daemon thread to the VM. This is useful when the VM should not wait for this thread to exit upon shutdown.
- Garbage collection policy: In addition to providing support for the large heaps enabled with a 64-bit address space, several subtle improvements were also made. One of the most important is in the area of generation sizing. J2SE version 1.4.0 provides improved heuristics for selecting the most efficient settings, which can have a dramatic effect on GC throughput.

Reliability, Availability, Serviceability (RAS)

- A command-line option, `-Xcheck:jni`, for performing additional JNI checks became available. This enables checks for argument validity to be run during development, where they can be detected before they are deployed and slow down production runs. Specifically, the Java HotSpot VM validates the parameters passed to the JNI function as well as the runtime environment data before processing the JNI request. Any invalid data encountered indicates a problem in the native code, and the VM will terminate with a fatal error in such cases.
- Java Platform Debugger Architecture (JPDA) Enhancements: JPDA is a multitiered debugging architecture that allows tool developers to easily create debugger applications which run portably across platforms, VM implementations, and SDK versions. Enhancements included in version 1.4:
  - Full Speed Debugging: In versions 1.3 and earlier of the Java HotSpot VM, when debugging was enabled, the program executed using only the interpreter. Full speed debugging offers the full performance advantage of HotSpot compiler technology to VM programs running with debugging enabled. The improved performance enables easier debugging of long running programs. As well, testing can proceed at full speed, and the launch of a debugger can occur on an exception.
  - HotSwap Class File Replacement: This feature provides the ability to substitute modified code in a running application through the debugger APIs. HotSwap adds functionality to the JPDA, enabling a class to be updated while under the control of a debugger. For example, developers can recompile a single class, and replace the old instance with a new instance.
  - Support For Debugging Other Languages: The JPDA was extended so that non Java language source code, which is translated into Java language source or Java VM class file format, can be debugged at the original non Java programming language source code level.
- Error-reporting mechanism: When the VM detects a crash in native code, such as JNI code written by the developer, or when the JVM itself crashes, it will print and log debug information about the crash. This error message normally will include information such as the function name, library name, source-file name, and line number where the error occurred. The result is that developers can more easily and efficiently debug their applications. If an error message indicates a problem in the JVM code itself, it allows a developer to submit a more accurate and helpful bug report.
- Solaris OE threading scalability improvements: Java HotSpot VM can take advantage of the new thread model available for the Solaris 8 and Solaris 9 OE. The thread model maps lightweight processes (LWPs) one-to-one to Solaris OE kernel threads. This model improves stability, decreases thread starvation, provides more balanced thread scheduling, and improves system-level synchronization and overall scalability.
  In a benchmark launching worker threads equal to the number of CPUs, all the threads perform nearly equal amounts of operations (within five percent); whereas with J2SE 1.3, the spread between threads doing the most and fewest operations could exceed 20 percent. This improves CPU utilization on systems with multiple CPUs using version 1.4.
- Signal-chaining facility: Signal-chaining enables the Java Platform to better interoperate with native code that installs its own signal handlers. The facility works on both Solaris OE and Linux platforms. The signal-chaining facility was introduced to remedy a problem with signal handling in previous versions of the Java HotSpot VM. Prior to version 1.4, the Java HotSpot VM would not allow application-installed signal handlers for certain signals including, for example, SIGBUS,

SIGSEGV, or SIGILL, since those signal handlers could conflict with the signal handlers used internally by the Java HotSpot VM.

Earlier Enhancements

The Java HotSpot VM introduced many advanced capabilities in version 1.3, and these capabilities are still part of the core functionality. The most notable include:

- Runtime
  - Ultra-fast thread synchronization, resulting in maximum performance of thread-safe Java technology-based programs.
  - Better fatal error reporting: When a fatal error occurs in the virtual machine, there is improved output, including a mechanism to detect if the crash was in the VM itself, or whether it was caused by user-supplied native code external to the VM.
- Compiler Optimizations
  - Range check elimination: The Java programming language specification requires array bounds checking to be performed with each array access. An index bounds check can be eliminated when the compiler can prove that an index used for an array access is within bounds.
  - Loop unrolling: The Server VM features loop unrolling, a standard compiler optimization that enables faster loop execution. Loop unrolling increases the loop body size while simultaneously decreasing the number of iterations. Loop unrolling also increases the effectiveness of other optimizations.
  - Instruction scheduling: Machine instructions generated by the compiler's code generator do not necessarily appear in optimal order for a particular hardware platform. Instruction scheduling is a compiler optimization that rearranges the generated machine instructions such that the execution speed is improved. This was a SPARC 3-specific optimization.
- Object-oriented optimizations for the Java reflection API: The compiler is aware of important library functions, which results in better performance when generating code for them.

Chapter 2

The Java HotSpot VM Architecture

Overview

The Java HotSpot Virtual Machine is Sun's VM for the Java platform. It delivers the optimal performance for Java applications using many advanced techniques, incorporating a state-of-the-art memory model, garbage collector, and adaptive optimizer. It is written in a high-level, object-oriented style, and features:

- Uniform object model
- Interpreted, compiled, and native frames all use the same stack
- Preemptive multithreading based on native threads
- Accurate generational and compacting garbage collection
- Ultra-fast thread synchronization
- Dynamic deoptimization and aggressive compiler optimizations
- System-specific runtime routines generated at VM startup time
- Compiler interface supporting parallel compilations
- Run-time profiling focuses compilation effort only on "hot" methods

The J2SE SDK version 1.4.1 release includes two flavors of the VM—a client-side offering, and a VM tuned for server applications. These two solutions share the Java HotSpot runtime environment code base, but use different compilers that are suited to the distinctly unique performance characteristics of clients and servers. These differences include the compilation inlining policy and heap defaults.

The J2SE SDK contains both of the these systems in the distribution, so developers can choose which system they want by specifying `-client` or `-server`.

Although the Server and the Client VMs are similar, the Server VM has been specially tuned to maximize peak operating speed. It is intended for executing long-running server applications, which need the fastest possible operating speed more than a fast start-up time or smaller runtime memory footprint.

The Client VM compiler serves as an upgrade for both the Classic VM and the just-in-time (JIT) compilers used by previous versions of the Java SDK. The Client VM offers improved run time performance for applications and applets. The Java HotSpot Client VM has been specially tuned to reduce application start-up time and memory footprint, making it particularly well suited for client environments. In general, the client system is better for GUIs.

The Client VM compiler does not try to execute many of the more complex optimizations performed by the compiler in the Server VM, but in exchange, it requires less time to analyze and

compile a piece of code. This means the Client VM can start up faster and requires a smaller memory footprint.

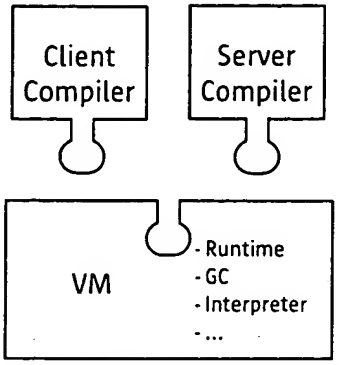

Figure 2-2: The Java HotSpot Client VM, on the left, and the Java HotSpot Server VM, on the right, use different compiler but otherwise interface to the same virtual machine, using the same garbage collection (GC) routine, interpreter, thread and lock subsystems, and so on.

The Server VM contains an advanced adaptive compiler that supports many of the same types of optimizations performed by optimizing C++ compilers, as well as some optimizations that cannot be done by traditional compilers, such as aggressive inlining across virtual method invocations. This is a competitive and performance advantage over static compilers. Adaptive optimization technology is very flexible in its approach, and typically outperforms even advanced static analysis and compilation techniques.

Both solutions deliver extremely reliable, secure, and maintainable environments to meet the demands of today's enterprise customers.

Memory Model

Handleless Objects

In previous versions of the Java virtual machine, such as the Classic VM, indirect handles are used to represent object references. While this makes relocating objects easier during garbage collection, it represents a significant performance bottleneck, because accesses to the instance variables of Java programming language objects require two levels of indirection.

In the Java HotSpot VM, no handles are used by Java code. Object references are implemented as direct pointers. This provides C-speed access to instance variables. When the object is relocated during memory reclamation, the garbage collector is responsible for finding and updating all references to the object in place.

Two-Word Object Headers

The Java HotSpot VM uses a two machine-word object header, as opposed to three words in the Classic VM. Since the average Java object size is small, this has a significant impact on space consumption—saving approximately eight percent in heap size for typical applications. The first header word contains information such as the identity hash code and GC status information. The second is a reference to the object's class. Only arrays have a third header field, for the array size.

Reflective Data are Represented as Objects

Classes, methods, and other internal reflective data are represented directly as objects on the heap (although those objects may not be directly accessible to Java technology-based programs). This not only simplifies the VM internal object model, but also allows classes to be collected by the same garbage collector used for other Java programming language objects.

Native Thread Support, Including Preemption and Multiprocessing

Per-thread method activation stacks are represented using the host operating system's stack and thread model. Both Java programming language methods and native methods share the same stack, allowing fast calls between the C and Java programming languages. Fully preemptive Java programming language threads are supported using the host operating system's thread scheduling mechanism.

A major advantage of using native OS threads and scheduling is the ability to take advantage of native OS multiprocessing support transparently. Because the Java HotSpot VM is designed to be insensitive to race conditions caused by preemption and/or multiprocessing while executing Java programming language code, the Java programming language threads will automatically take advantage of whatever scheduling and processor allocation policies the native OS provides.

Garbage Collection

The generational nature of the HotSpot memory system provides the flexibility to use specific garbage collection algorithms suited to the needs of a diverse set of applications. In addition to the garbage collection algorithms available in the previous version of the Java HotSpot VM, two new collectors are available as options in this release: a multithreaded synchronous copying collector for use in the young generation, and a "mostly concurrent" asynchronous non-copying mark-sweep collector best suited for the old generation.

Background

A major attraction of the Java programming language for programmers is that it is the first mainstream programming language to provide built-in automatic memory management, or garbage collection (GC). In traditional languages, dynamic memory is allocated using an explicit allocate/free model. In practice, this turns out to be not only a major source of memory leaks, program bugs, and crashes in programs written in traditional languages, but also a performance bottleneck and a major impediment to modular, reusable code. (Determining free points across module boundaries is nearly impossible without explicit and hard-to-understand cooperation between modules.) In the Java programming language, garbage collection is also an important part of the "safe" execution semantics required to support the security model.

A garbage collector automatically handles *freeing* of unused object memory behind the scenes by reclaiming an object only when it can prove that the object is no longer accessible to the running program. Automation of this process completely eliminates not only the memory leaks caused by freeing too little, but also the program crashes and hard-to-find reference bugs caused by freeing too much.

Traditionally, garbage collection has been considered an inefficient process that impeded performance, relative to an explicit-free model. In fact, with modern garbage collection technology, performance has improved so much that overall performance is actually substantially better than that provided by explicit freeing of objects.

The Java HotSpot Garbage Collector

In addition to including the state-of-the-art features described below, the memory system is designed as a clean, object-oriented framework that can easily be instrumented, experimented with, or extended to use new garbage collection algorithms.

The major features of the Java HotSpot garbage collector are presented below. Overall, these capabilities are well-suited both for applications where the highest possible performance is

needed, and for long-running applications where memory leaks and memory inaccessibility due to fragmentation are highly undesirable.

Accuracy

The Java HotSpot garbage collector is a fully accurate collector. In contrast, many other garbage collectors are conservative or partially accurate. While conservative garbage collection can be attractive because it is very easy to add to a system without garbage collection support, it has certain drawbacks. In general, conservative garbage collectors are prone to memory leaks, disallow object migration, and can cause heap fragmentation.

A conservative collector does not know for sure where all object references are located. As a result, it must be conservative by assuming that memory words that appear to refer to an object are in fact object references. This means that it can make certain kinds of mistakes, such as confusing an integer for an object pointer. Memory cells that look like a pointer are regarded as a pointer—and GC becomes inaccurate. This has several negative impacts. First, when such mistakes are made (which in practice is not very often), memory leaks can occur unpredictably in ways that are virtually impossible for application programmers to reproduce or debug. Second, since it might have made a mistake, a conservative collector must either use handles to refer indirectly to objects—decreasing performance—or avoid relocating objects, because relocating handleless objects requires updating all references to the objects. This cannot be done if the collector does not know for sure whether an apparent reference is a real reference. The inability to relocate objects causes object memory fragmentation and, more importantly, prevents use of the advanced generational copying collection algorithms described below.

Because the Java HotSpot collector is fully accurate, it can make several strong design guarantees that a conservative collector cannot make:

- All inaccessible object memory can be reclaimed reliably.
- All objects can be relocated, allowing object memory compaction, which eliminates object memory fragmentation and increases memory locality.

An accurate garbage collection mechanism avoids accidental memory leaks, enables object migration, and provides for full heap compaction. The GC mechanism in the Java Hotspot VM scales well to very large heaps.

Generational Copying Collection

The Java HotSpot VM employs a state-of-the-art generational copying collector, which provides two major benefits:

- Increased allocation speed and overall garbage collection efficiency for most programs, compared to nongenerational collectors
- Corresponding decrease in the frequency and duration of user-perceivable garbage collection pauses

A generational collector takes advantage of the fact that in most programs, the vast majority of objects (often greater than 95 percent) are very short lived (for example, they are used as temporary data structures). By segregating newly created objects into an object nursery, a generational collector can accomplish several things. First, because new objects are allocated contiguously in stack-like fashion in the object nursery, allocation becomes extremely fast, since it merely involves updating a single pointer and performing a single check for nursery overflow. Secondly, by the time the nursery overflows, most of the objects in the nursery are already dead,

allowing the garbage collector to simply move the few surviving objects elsewhere, and avoid doing any reclamation work for dead objects in the nursery.

Parallel Copying Collector

The single-threaded copying collector described above, while suitable for many deployments, could become a bottleneck to scaling in an application that is otherwise parallelized to take advantage of multiple processors. To take full advantage of all available CPUs on a multiprocessor machine, version 1.4.1 of the HotSpot JVM offers an optional multithreaded collector for the young generation, in which the tracing and copying of live objects is accomplished by multiple threads working in parallel. The implementation has been carefully tuned to balance the collection work between all available processors, allowing the collector to scale up to large numbers of processors. This reduces the pause times for collecting young space and maximizes garbage collection throughput. The parallel collector has been tested with systems containing more than 100 CPU's and 0.5 terabytes of heap.

When moving objects, the parallel collector tries to keep related objects together, resulting in improved memory locality and cache utilization, and leading to improved mutator performance. This is accomplished by copying objects in *depth first order.*

The parallel collector also uses available memory more optimally. It does not need to keep a portion of the old object space in reserve to guarantee space for copying all live objects. Instead, it uses a novel technique to speculatively attempt to copy objects. If old object space is scarce this technique allows the collector to switch smoothly to compacting the heap without the need for holding any space in reserve. This results in better utilization of the available heap space.

Finally, the parallel collector is able to dynamically adjust its tunable parameters in response to the application's heap allocation behavior, leading to improved garbage collection performance over a wide range of applications and environments. This means less hand-tuning work for customers. This capability is currently available only for the parallel collector, but is expected to be further refined and extended to other collectors in future releases.

In comparison with the default single-threaded collector, the break-even point for the parallel collector appears to be somewhere between two and four CPUs, depending on the platform and the application. This is expected to further improve in future releases.

As mentioned above, this collector parallelizes only the work for minor (young space) collections. Major (old space) collections are also expected to be parallelized in future releases.

Mark-Compact Old Object Collector

Although the generational copying collector collects most dead objects efficiently, longer-lived objects still accumulate in the old object memory area. Occasionally, based on low-memory conditions or programmatic requests, an old object garbage collection must be performed. The Java HotSpot VM can use a standard mark-compact collection algorithm, which traverses the entire graph of live objects from its *roots*, then sweeps through memory, compacting away the gaps left by dead objects. By compacting gaps in the heap, rather than collecting them into a freelist, memory fragmentation is eliminated, and old object allocation is streamlined by eliminating freelist searching.

Incremental Low-Pause Garbage Collector

The mark-compact collector does not eliminate all user-perceivable pauses. User-perceived GC pauses occur when old objects (objects that have lived for awhile in machine terms) need to be

garbage collected, and these pauses are proportional to the amount of live object data that exists. This means that the pauses can become arbitrarily large as more data is manipulated, which is a very undesirable property for server applications, animation, or other soft-real time applications.

To solve this problem, the Java HotSpot VM provides an alternative old space garbage collector that is fully incremental, virtually eliminating user-detectable garbage collection pauses. This incremental collector scales smoothly, providing relatively constant pause times, even when extremely large object data sets are being manipulated. This provides excellent behavior for:

- Server applications, especially high-availability applications
- Applications that manipulate very large live object data sets
- Applications where all user-noticeable pauses are undesirable, such as games, animation, or other highly interactive applications

The low-pause collector works by using an incremental old space collection scheme referred to as the *train* algorithm. This algorithm breaks up old space collection pauses into many tiny pauses (typically less than ten milliseconds) that can be spread out over time so that the user never notices a pause. Since the train algorithm is not a hard-real time algorithm, it cannot guarantee an upper limit on pause times. However, in practice much larger pauses are extremely rare, and are not caused directly by large data sets.

The low-pause collector also has the highly desirable side benefit of improving memory locality. This happens because the algorithm attempts to relocate groups of tightly coupled objects into regions of adjacent memory, providing excellent paging and cache locality properties for those objects. This can also benefit highly multithreaded applications that operate on distinct sets of object data.

Mostly Concurrent Mark-Sweep Collector

For applications that require large heaps, collection pauses induced by the default old generation mark-compact collector can often cause disruptions, as application threads are paused for a period that is proportional to the size of the heap. Version 1.4.1 of the Java HotSpot VM has implemented an optional concurrent collector for the old object space that can take advantage of spare processor cycles (or spare processors) to collect large heaps while pausing the application threads for very short periods. This is accomplished by doing the bulk of the tracing and sweeping work while the application threads are executing. In some cases, there may be a small decline in peak application throughput as some processor cycles are devoted to concurrent collection activity; however, both average- and worst-case garbage collection pause times are often reduced by one or two orders of magnitude, allowing much smoother application response without the burstiness sometimes seen when the default synchronous mark-compact algorithm operates on large heaps.

64-bit Architecture

Previous releases of the Java HotSpot VM were limited to addressing four gigabytes of memory—even on 64-bit operating systems such as the Solaris OE. While four gigabytes is a lot for a desktop system, modern servers can contain far more memory. For example, the new Sun Fire™ 15K server comes in a standard configuration with 288 gigabytes of memory. With a 64-bit JVM, Java technology-based applications can now utilize the full memory of such a system.

There are several classes of applications where using 64-bit addressing can be useful. For example, those that store very large data sets in memory. Applications can now avoid the overhead of paging data from disk or extracting it from a an RDBMS. This can lead to dramatic performance improvements in applications of this type.

The Java HotSpot Server VM is now 64-bit safe, and the Server VM includes support for both 32-bit and 64-bit operations. Users can select either 32-bit or 64-bit operation by using command-line flags –d32 or –d64, respectively. Users of the Java Native Interface will need to recompile their code to run it on the 64-bit VM.

Object Packing

Object packing functionality has been added to minimize the wasted space between data types of different sizes. This is primarily a benefit in 64-bit environments, but offers a small advantage even in 32-bit VMs.

For example:

```
public class Button {
  char shape;
  String label;
  int xposition;
  int yposition;
  char color;
  int joe;
  object mike;
  char armed;
}
```

This would waste space between:

`color` and `joe` (three bytes to pad to an `int` boundary)

`joe` and `mike` (four bytes on a 64-bit VM to pad to a pointer boundary)

Now, the fields are reordered to look like this:

```
...
  object mike;
  int joe;
  char color;
  char armed;
...
```

In this example, no memory space is wasted.

Ultra-Fast Thread Synchronization

The Java programming language allows for use of multiple, concurrent paths of program execution—*threads*. The Java programming language provides language-level thread synchronization, which makes it easy to express multithreaded programs with fine-grained locking. Previous synchronization implementations were highly inefficient relative to other micro-operations in the Java programming language, making use of fine-grain synchronization a major performance bottleneck.

The Java HotSpot VM incorporates a breakthrough in thread synchronization implementation that boosts synchronization performance by a large factor. As a result, synchronization performance becomes so fast that it is not a significant performance issue for the vast majority of real-world programs.

In addition to the space benefits mentioned in the Memory Model section, the synchronization mechanism delivers its performance benefits by providing ultra-fast, constant-time performance for all uncontended synchronizations, which dynamically comprise the majority of synchronizations.

The Java HotSpot VM provides a leaner, speedier thread-handling capability that is designed to scale readily for use in large, shared-memory multiprocessor servers.

New I/O APIs

The new I/O (NIO) APIs introduced in version 1.4 provide new features and improved performance in the areas of buffer management, scalable network and file I/O, character-set support, and regular-expression matching. These new APIs are supported in both client and server compilers. With NIO, developers can now write ultra-scalable, high-performance server applications such as Web, application, file, and database servers. They can also write compute-intensive scientific, technical, and graphics applications that require fast access to large quantities of data. I/O operations that previously required programming in C or C++ can now be performed using the Java language, but with the performance of a native C or C++ application.

- The new network I/O package dramatically increases the number of simultaneous connections that a server can handle by removing the need to dedicate one thread to every open connection.
- New file I/O supports read, write, copy, and transfer operations that are up to twice as fast as the current file I/O facilities. It also supports file locking, memory-mapped files, and multiple concurrent read/write operations.

Chapter 3

Debugging Support

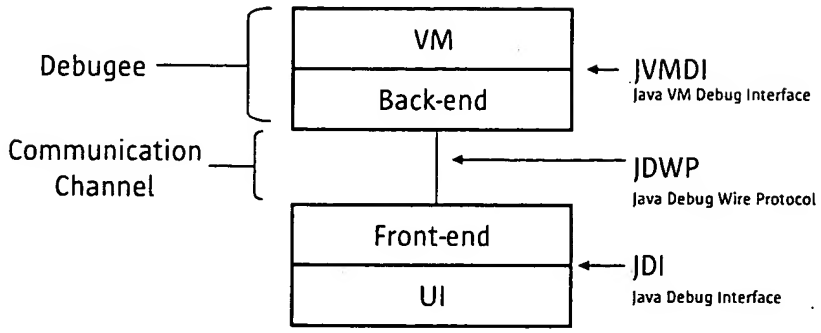

Figure 3-3: The Java Platform Debugger Architecture Structure

Overview

The Java Platform Debugger Architecture (JPDA) consists of two interfaces, the Java VM Debug Interface (JVMDI) and the Java Debug Interface (JDI), a protocol (JDWP), and two software components which tie them together (back-end and front-end). The intent is of this architecture is to:

- Provide standard interfaces which enable Java debugging tools to be easily written without regard to platform specifics such as hardware, operating system, and virtual machine implementation.
- Describe a complete architecture for implementing these interfaces, including remote and cross-platform debugging.
- Provide a reference implementation of this architecture.
- Provide a highly modular architecture where the implementation or client of an interface can be different than the reference implementation, or different from the JPDA component.

Java Virtual Machine Debugger Interface (JVMDI)

The JVMDI describes the functionality provided by a VM to enable debugging of Java programming language applications running under this VM. In the reference implementation of JPDA, JVMDI is implemented by the Java HotSpot VM.

The JVMDI defines the services a VM must provide for debugging. The JVMDI includes requests for information (for example, current stack frame), actions (set a breakpoint), and notification (when a breakpoint has been hit). A debugger may make use of VM information other than this, such as the Java Native Interface (JNI), but the JVMDI is the source of all debugger specific information.

Specifying the VM Interface allows any VM implementor to plug easily into the debugging architecture. It also allows alternate communication channel implementations. VM implementations that do not adhere to this interface can still provide access via the Java Debug Wire Protocol (JDWP).

Java Debug Wire Protocol (JDWP)

The JDWP defines the format of information and requests transferred between the process being debugged and the debugger front-end, which implements the Java Debug Interface. It does not define the transport mechanism—socket, serial line, shared memory, and so on.

The specification of the protocol allows the process being debugged and debugger front-end to run under separate VM implementations and on separate platforms. It also enables the front-end to be written in a language other than Java, or the debuggee to be non-native (such as Java technology). Information and requests are roughly at the level of the JVMDI, but will include additional information and requests necessitated by bandwidth issues, such as information filtering and batching.

In the reference implementation, a module called the back-end runs as an agent of the HotSpot VM and receives JDWP packets from the debugger front-end. This back-end code executes the commands it receives over the JDWP, calling into the HotSpot VM via JVMDI when necessary. Results are returned back to the debugger front-end via JDWP.

Java Debug Interface (JDI)

The JDI provides a pure Java programming language interface for debugging applications based on the Java language. The JDI is a high-level Java API providing functionality that is needed by debuggers and similar systems to access and control the state of the virtual machine.

In JPDA, the JDI offers a remote view in the debugger process of a virtual machine in the debuggee process. It is implemented by the front-end (above) while a debugger-like application (IDE, debugger, tracer, monitoring tool, and so on) is the client.

The JDI defines information and requests at the user code level. It provides introspective access to a running virtual machine's state, class, array, interface, and primitive types, plus instances of those types. The JDI also provides explicit control over a virtual machine's execution. JDI is the highest layer of the Java Platform Debugger Architecture (JPDA).

Full-Speed Debugging

The Java HotSpot VM now uses *full-speed debugging*. In previous version of the VM, when debugging was enabled, the program executed using only the interpreter. Now, the full performance advantage of HotSpot technology is available to programs, even with compiled code. The improved performance allows long-running programs to be more easily debugged. It also allows testing to proceed at full speed. Once there is an exception, the debugger launches with full visibility to code sources.

HotSwap Class File Replacement

This new feature encapsulates the ability to substitute modified code in a running application through the debugger APIs. For example, a developer can recompile a single class and replace the old instance with the new instance.

HotSwap adds functionality to the JPDA, allowing a class to be updated while under the control of a debugger. The two central components of this functionality are `RedefineClasses` which replaces the class definitions, and `PopFrame` which pops frames off the stack, allowing a method that has been redefined to be reexecuted.

In the reference implementation, this functionality is implemented at the JVMDI layer and made available through the higher layers of JPDA - the JDWP and the JDI.

VMDeathRequests

Using class `VMDeathRequest`, a request can be made for notification when the target VM terminates. When an enabled `VMDeathRequest` is satisfied, an `EventSet` containing a `VMDeathEvent` will be placed on the `EventQueue`.

This request would typically be created so that a `VMDeathEvent` with a suspend policy of `SUSPEND_ALL` will be sent. This event can be used to assure completion of any processing that requires the VM to be alive (such as event processing). Even without creating a `VMDeathRequest`, a single unsolicited `VMDeathEvent` will be sent with a suspend policy of `SUSPEND_NONE`.

Logging

A new logging facility has been added to log garbage collection events from the virtual machine. The new `-Xloggc:file` option reports on each garbage collection event, as with `-verbose:gc`, but logs this data to file. In addition to the information `-verbose:gc` provides, each reported event is preceded by the time (in seconds) since the first garbage collection event.

Chapter 4

The Java HotSpot Compilers

Overview

Most attempts to accelerate Java programming language performance have focused on applying compilation techniques developed for traditional languages. Just-in-time (JIT) compilers are essentially fast traditional compilers that translate the Java technology bytecodes into native machine code on the fly. A JIT running on the end user's machine actually executes the bytecodes and compiles each method the first time it is executed.

However, there are several issues with JIT compilation. First, because the compiler runs on the execution machine in *user time*, it is severely constrained in terms of compile speed: if it is not very fast, then the user will perceive a significant delay in the startup of a program or part of a program. This entails a trade-off that makes it far more difficult to perform advanced optimizations, which usually slow down compilation performance significantly.

Secondly, even if a JIT had time to perform full optimization, such optimizations are less effective for the Java programming language than for traditional languages like C and C++. There are a number of reasons for this:

- The Java language is dynamically *safe*, meaning that it ensures that programs do not violate the language semantics or directly access unstructured memory. This means dynamic type-tests must frequently be performed (when casting, and when storing into object arrays).
- The Java language allocates all objects on the *heap*, in contrast to C++, where many objects are stack allocated. This means that object allocation rates are much higher for the Java language than for C++. In addition, because the Java language is garbage collected, it has very different types of memory allocation overhead (including potentially scavenging and write-barrier overhead) than C++.
- In the Java language, most method invocations are *virtual* (potentially polymorphic), and are more frequently used than in C++. This means not only that method invocation performance is more dominant, but also that static compiler optimizations (especially global optimizations such as inlining) are much harder to perform for method invocations. Many traditional optimizations are most effective between calls, and the decreased distance between calls in the Java language can significantly reduce the effectiveness of such optimizations, since they have smaller sections of code to work with.
- Java technology-based programs can change on the fly due to a powerful ability to perform dynamic loading of classes. This makes it far more difficult to perform many types of global optimizations. The compiler must not only be able to detect when these optimizations become

invalid due to dynamic loading, but also be able to undo or redo those optimizations during program execution, even if they involve active methods on the stack. This must be done without compromising or impacting Java technology-based program execution semantics in any way.

As a result, any attempt to achieve fundamental advances in Java language performance must provide nontraditional answers to these performance issues, rather than blindly applying traditional compiler techniques.

The Java HotSpot VM architecture addresses the Java language performance issues described above by using adaptive optimization technology.

Hot Spot Detection

Adaptive optimization solves the problems of JIT compilation by taking advantage of an interesting program property. Virtually all programs spend the vast majority of their time executing a minority of their code. Rather than compiling method by method, just in time, the Java HotSpot VM immediately runs the program using an interpreter, and analyzes the code as it runs to detect the critical hot spots in the program. Then it focuses the attention of a global native-code optimizer on the hot spots. By avoiding compilation of infrequently executed code (most of the program), the Java HotSpot compiler can devote more attention to the performance-critical parts of the program, without necessarily increasing the overall compilation time. This hot spot monitoring is continued dynamically as the program runs, so that it literally adapts its performance on the fly to the user's needs.

A subtle but important benefit of this approach is that by delaying compilation until after the code has already been executed for a while (measured in machine time, not user time), information can be gathered on the way the code is used, and then utilized to perform more intelligent optimization. As well, the memory footprint is decreased. In addition to collecting information on hot spots in the program, other types of information are gathered, such as data on caller-callee relationships for virtual method invocations.

Method Inlining

The frequency of virtual method invocations in the Java programming language is an important optimization bottleneck. Once the Java HotSpot adaptive optimizer has gathered information during execution about program hot spots, it not only compiles the hot spot into native code, but also performs extensive method inlining on that code.

Inlining has important benefits. It dramatically reduces the dynamic frequency of method invocations, which saves the time needed to perform those method invocations. But even more importantly, inlining produces much larger blocks of code for the optimizer to work on. This creates a situation that significantly increases the effectiveness of traditional compiler optimizations, overcoming a major obstacle to increased Java programming language performance.

Inlining is synergistic with other code optimizations, because it makes them more effective. As the Java HotSpot compiler matures, the ability to operate on large, inlined blocks of code will open the door to a host of even more advanced optimizations in the future.

Dynamic Deoptimization

Although inlining, described in the last section, is an important optimization, it has traditionally been very difficult to perform for dynamic object-oriented languages like the Java language. Furthermore, while detecting hot spots and inlining the methods they invoke is difficult enough, it is still not sufficient to provide full Java programming language semantics. This is because programs

written in the Java language can not only change the patterns of method invocation on the fly, but can also dynamically load new Java code into a running program.

Inlining is based on a form of global analysis. Dynamic loading significantly complicates inlining, because it changes the global relationships in a program. A new class may contain new methods that need to be inlined in the appropriate places. So the Java HotSpot VM must be able to dynamically deoptimize (and then reoptimize, if necessary) previously optimized hot spots, even while executing code for the hot spot. Without this capability, general inlining cannot be safely performed on Java technology-based programs.

Java HotSpot Client Compiler

The Java HotSpot Client Compiler is a simple, fast three-phase compiler. In the first phase, a platform-independent front end constructs a high-level intermediate representation (HIR) from the bytecodes. In the second phase, the platform-specific back end generates a low-level intermediate representation (LIR) from the HIR.

The final phase does peephole optimization on the LIR and generates machine code from it. Emphasis is placed on extracting and preserving as much information as possible from the bytecodes. It focuses on local code quality and does very few global optimizations, since those are often the most expensive in terms of compile time. It supports inlining any function that has no exception handlers or synchronization, and also supports deoptimization for debugging and inlining.

Java HotSpot Server Compiler

The server compiler is tuned for the performance profile of typical server applications. The Java HotSpot Server Compiler is a high-end fully optimizing compiler. It uses an advanced static single assignment (SSA)-based IR for optimizations. The optimizer performs all the classic optimizations, including dead code elimination, loop invariant hoisting, common subexpression elimination, and constant propagation. It also features optimizations more specific to Java technology, such as null-check and range-check elimination. The register allocator is a global graph coloring allocator and makes full use of large register sets that are commonly found in RISC microprocessors. The compiler is highly portable, relying on a machine description file to describe all aspects of the target hardware. While the compiler is slow by JIT standards, it is still much faster than conventional optimizing compilers. And the improved code quality pays back the compile time by reducing execution times for compiled code. The server compiler performs full inlining and full deoptimization.

Chapter 5

Impact on Software Reusability

Overview

A primary benefit of object-oriented programming is that it can increase development productivity by providing powerful language mechanisms for software reuse. In practice however, such reusability is rarely attained. Extensive use of these mechanisms can significantly reduce performance, which leads programmers to use them sparingly. A surprising side effect of the Java HotSpot technology is that it significantly reduces this performance cost. Sun believes this will have a major impact on how object-oriented software is developed, by allowing companies to take full advantage of object-oriented reusability mechanisms for the first time, without compromising the performance of their software.

Examples of this effect are easy to come by. A survey of programmers using the Java programming language will quickly reveal that many avoid using fully virtual methods (and also write bigger methods), because they believe that every virtual method invocation entails a significant performance penalty. Ubiquitous, fine-grain use of virtual methods, such as methods that are not *static* or *final* in the Java programming language, is extremely important to the construction of highly reusable classes, because each such method acts as a *hook* that allows new subclasses to modify the behavior of the superclass.

Because the Java HotSpot VM can automatically inline the vast majority of virtual method invocations, this performance penalty is dramatically reduced, and in many cases, eliminated altogether.

It is hard to overstate the importance of this effect. It has the potential to fundamentally change the way object-oriented code is written, since it significantly changes the performance trade-offs of using important reusability mechanisms. In addition, it has become clear that as object-oriented programming matures, there is a trend towards both finer-grain objects and finer-grain methods. This trend points strongly to increases in the frequency of virtual method invocations in the coding styles of the future. As these higher-level coding styles become prevalent, the advantages of Java HotSpot technology will become even more pronounced.

Chapter 6

Summary

The Java HotSpot VM delivers optimal performance for Java applications, delivering advanced optimization, garbage collection, and thread synchronization capabilities. In addition, this latest release offers new debugging capabilities designed to improve overall reliability and availability of Java technology-based applications. The Java HotSpot VM provides separate compilers for client and server environments so that applications can be optimized according to their target deployment environments. Scalability has been significantly enhanced with the availability of a 64-bit Java HotSpot Server Compiler.

With the Java HotSpot VM, client applications start up faster and require a smaller memory footprint, while server applications can achieve better sustained performance over the long run. Both solutions deliver an extremely reliable, secure, and maintainable environment to meet the demands of today's enterprise customers.

Chapter 7

Availability

The Java HotSpot VM is included in the J2SE v1.4.1 platform environment. It is available at *java.sun.com* for the following environments:

- Solaris Operating Environment for SPARC platforms. To run the 64-bit Java HotSpot VM, the optional Solaris OE 64-bit package must be installed.
- Solaris Operating Environment for Intel platforms
- Linux Operating Systems: Red Hat versions 7.1 and 6.2 are the officially supported platforms. Most testing of J2SE SDK 1.4.1 for Linux has been conducted on Red Hat 7.1 with the sawfish window manager and Red Hat 6.2 with the Gnome desktop, and these are the officially supported platforms. However, J2SE SDK 1.4.1 has undergone limited testing on other Linux operating systems. See *java.sun.com/j2se/1.4/install-linux.html* for more information.
- Microsoft Windows 95, 98, NT 4.0, ME, XP Home, XP Professional, 2000 Professional, 2000 Server, or 2000 Advanced Server operating systems running on Intel hardware.

The Java HotSpot Server VM is also included in the release of Java 2 technology from Hewlett-Packard for their PA-RISC hardware platforms. The Java HotSpot Client VM v1.31 is shipping as part of Apple's Mac OS X.

Chapter 8

Resources

These Web sites provide additional information:

Garbage collection: java.sun.com/docs/hotspot/gc/index.html

Threading models: java.sun.com/docs/hotspot/threads/threads.html

Large heaps: java.sun.com/docs/hotspot/ism.html

Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, CA 94303-4900 USA Phone 800 786-7638 or +1 512 434-1577 Web sun.com

Sun Worldwide Sales Offices: Africa (North, West and Central) +33-13-067-4680, Argentina +5411-4317-5600, Australia +61-2-9844-5000, Austria +43-1-60563-0, Belgium +32-2-704-8000, Brazil +55-11-5187-2100, Canada +905-477-6745, Chile +56-2-3724500, Colombia +571-629-2323, Commonwealth of Independent States +7-502-935-8411, Czech Republic +420-2-3300-9311, Denmark +45 4556 5000, Egypt +202-570-9442, Estonia +372-6-308-900, Finland +358-9-525-561, France +33-134-03-00-00, Germany +49-89-46008-0, Greece +30-1-618-8111, Hungary +36-1-489-8900, Iceland +354-563-3010, India-Bangalore +91-80-2298989/2295454; New Delhi +91-11-6106000; Mumbai +91-22-697-8111, Ireland +353-1-8055-666, Israel +972-9-9710500, Italy +39-02-641511, Japan +81-3-5717-5000, Kazakhstan +7-3272-466774, Korea +822-2193-5114, Latvia +371-750-3700, Lithuania +370-729-8468, Luxembourg +352-49 11 33 1, Malaysia +603-21161888, Mexico +52-5-258-6100, The Netherlands +00-31-33-45-15-000, New Zealand-Auckland +64-9-976-6800; Wellington +64-4-462-0780, Norway +47 23 36 96 00, People's Republic of China-Beijing +86-10-6803-5588; Chengdu +86-28-619-9333; Guangzhou +86-20-8755-5900; Shanghai +86-21-6466-1228; Hong Kong +852-2202-6688, Poland +48-22-8747800, Portugal +351-21-4134000, Russia +7-502-935-8411, Singapore +65-6438-1888, Slovak Republic +421-2-4342-94-85, South Africa +27 11 256-6300, Spain +34-91-596-9900, Sweden +46-8-631-10-00, Switzerland-German 41-1-908-90-00; French 41-22-999-0444, Taiwan +886-2-8732-9933, Thailand +662-344-6888, Turkey +90-212-335-22-00, United Arab Emirates +9714-3366333, United Kingdom +44-1-276-20444, United States +1-800-555-9SUN or +1-650-960-1300, Venezuela +58-2-905-3800